Today’s Outline

- **Admin:**
  - Final Exam – Tuesday December 11th, topic list posted soon
  - HW #6 – Sorting, due Thurs December 6 at 11pm

- **Sorting**
  - In-place and Stable Sorting

- **Dictionaries**
  - B-Trees

Trees so far

- **BST**

- **AVL**
CPU (has registers)

Time to access:
1 ns per instruction

Cache 2-10 ns

Main Memory 40-100 ns

Disk a few milliseconds (5-10 Million ns)

SRAM 8KB - 4MB

DRAM up to 10GB

many GB

---

$M$-ary Search Tree

• Maximum branching factor of $M$
• Complete tree has height =

# disk accesses for $find$:

Runtime of $find$:

---

Solution: B-Trees

• specialized $M$-ary search trees
• Each node has (up to) $M$-1 keys:
  – subtree between two keys $x$ and $y$ contains leaves with values $v$ such that
    $x \leq v < y$
• Pick branching factor $M$ such that each node takes one full 
  (page, block) of memory
B-Trees

What makes them disk-friendly?

1. **Many keys stored in a node**
   - All brought to memory/cache in one access!

2. **Internal nodes contain only keys**;
   - **Only leaf nodes contain keys and actual data**
   - The tree structure can be loaded into memory irrespective of data object size
   - Data actually resides in disk

---

**B-Tree: Example**

B-Tree with $M = 4$ (# *pointers* in internal node) and $L = 4$ (# *data items* in *leaf*)

---

**B-Tree Properties**

- Data is stored at the leaves
- All leaves are at the same depth and contain between $\lceil L/2 \rceil$ and $L$ data items
- Internal nodes store up to $M-1$ keys
- Internal nodes have between $\lceil M/2 \rceil$ and $M$ children
- Root (special case) has between 2 and $M$ children (or root could be a leaf)

---

†These are technically B*-Trees
Example, Again

B-Tree with \( M = 4 \) and \( L = 4 \)

B-trees vs. AVL trees

Suppose we have 100 million items (100,000,000):

• Depth of AVL Tree
• Depth of B+ Tree with \( M = 128 \), \( L = 64 \)

Building a B-Tree

The empty B-Tree

\( M = 3 \), \( L = 2 \)

Insert(3) \( \rightarrow \) Insert(14) \( \rightarrow \) Insert(10)

Now, Insert(1)?
Splitting the Root

Too many keys in a leaf!

And create a new root.

So, split the leaf.

Insert(1)

Overflowing leaves

Too many keys in a leaf!

So, split the leaf.

And add a new child.

Insert(59)

Propagating Splits

Too many keys in a leaf!

Split the leaf, but no space in parent!

Create a new root.

Add new child.

So, split the node.
### Insertion Algorithm

1. Insert the key in its leaf
2. If the leaf ends up with $L+1$ items, **overflow**!
   - Split the leaf into two nodes:
     - original with $\lceil (L+1)/2 \rceil$ items
     - new one with $\lfloor (L+1)/2 \rfloor$ items
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items, **overflow**!

3. If an **internal node** ends up with $M+1$ items, **overflow**!
   - Split the node into two nodes:
     - original with $\lceil (M+1)/2 \rceil$ items
     - new one with $\lfloor (M+1)/2 \rfloor$ items
   - Add the new child to the parent
   - If the parent ends up with $M+1$ items, **overflow**!

4. **Split** an overflowed root in two and hang the new nodes under a new root

   This makes the tree deeper!

---

### After More Routine Inserts

#### Insert(89)

#### Insert(79)

---

### Deletion

1. Delete item from leaf
2. Update keys of ancestors if necessary

#### Delete(59)

What could go wrong?
Deletion and Adoption

A leaf has too few keys!

So, borrow from a sibling

Does Adoption Always Work?

- What if the sibling doesn’t have enough for you to borrow from?

  e.g. you have \( \lceil L/2 \rceil - 1 \) and sibling has \( \lfloor L/2 \rfloor \)?

Deletion and Merging

A leaf has too few keys!

And no sibling with surplus!

But now an internal node has too few subtrees!

So, delete the leaf
Deletion with Propagation (More Adoption)

A Bit More Adoption

Pulling out the Root
The root has just one subtree!

Simply make the one child the new root!

Deletion Algorithm

1. Remove the key from its leaf

2. If the leaf ends up with fewer than \([L/2]\) items, underflow!
   - Adopt data from a sibling; update the parent
   - If adopting won’t work, delete node and merge with neighbor

3. If an internal node ends up with fewer than \([M/2]\) items, underflow!
   - Adopt from a neighbor; update the parent
   - If adoption won’t work, merge with neighbor
   - If the parent ends up with fewer than \([M/2]\) items, underflow!

4. If the root ends up with only one child, make the child the new root of the tree
   - This reduces the height of the tree!
Thinking about B-Trees

- B-Tree insertion can cause (expensive) splitting and propagation
- B-Tree deletion can cause (cheap) adoption or (expensive) deletion, merging and propagation
- Propagation is rare if $M$ and $L$ are large \( \text{(Why?)} \)
- If $M = L = 128$, then a B-Tree of height 4 will store at least 30,000,000 items

Tree Names You Might Encounter

FYI:
- B-Trees with $M = 3$, $L = x$ are called 2-3 trees
  - Nodes can have 2 or 3 pointers
- B-Trees with $M = 4$, $L = x$ are called 2-3-4 trees
  - Nodes can have 2, 3, or 4 pointers

Determining $M$ and $L$ for a B-Tree

1 page on disk = 1 KByte
Key = 8 bytes, Pointer = 4 bytes
Data = 256 bytes per record (includes key)

$M = \_\_\_\_\_\_\_\_\_\_\_\$

$L = \_\_\_\_\_\_\_\_\_\_\_\$

11/30/2012